

UNIVERSITY OF GLASGOW

Computing Department

SOFTWARE IMPLEMENTATION LANGUAGES FOR THE MULTUM COMPUTER MEMO NO. 2

Software Implementation Languages Memo No. 2
for the MULTUM Computer.

1. Introduction

This note deals with the question of the choice of language in which systems software for the MULTUM computer should be written at Glasgow University Computing Department (and hopefully elsewhere). The main recommendation of this note is that the cost-effective choice is the high-level language PASCAL [1] designed and implemented by Wirth [2] initially for the CDC6400. PASCAL is now also available on suitably - sized 1900 - series machines [3] and a /360 implementation is under development at Stanford. This note deals with

- (a) some possible alternative choices to PASCAL, together with the relative merits of these choices,
- and (b) the features which a MULTUM implementation of PASCAL should possess.

The MULTUM computer is currently the best example of a rapidly emerging class of well-designed machines of physically small size and low cost, but considerable power. Large MULTUM ALP/3 configurations are from the hardware point of view highly competitive with the machines touted by the major manufacturers for what is the bulk of their market (how many /360 30s, 40s and 50s has IBM sold?). These manufacturers, however, have learned that an important selling point for their machines is the provision of extensive (and expensive) software which enables their machines to be exploited satisfactorily, using the skills they expect their customers to have. The technical merits, or otherwise, of the available software have been extensively debated in the literature, so even a summary is pointless here. Three points do arise, however

- (i) the currently available software for currently available machines has been provided at greater expense, and with, generally speaking, a less satisfactory result than need be the case with the current state-of-the-art (which has been established through this effort!)
- (ii) there is considerable financial pressure to encourage manufacturers to stick to decisions about, and the technology of, software which represents an earlier state-of-the-art than the present,
- and (iii) it is difficult to see how major manufacturers can resist, at least in the short-to-medium term, an aggressive challenge from MULTUM-type computers with software of a comparable quality.

The above is particularly true for operating systems.

The intended bias of the preceding comments is towards the suggestion that well-designed system software developed with the best available software development tools is centrally important to the effective promotion/

promotion of the large (and also not-so-large) MULTUM machines outside the process-control market.

An important feature of the MULTUM ALP/3 architecture is the organization of the different addressing modes. This structure is particularly effective for the implementation of sophisticated data structures, and in particular for the effective implementation of the storage structures for block-structured languages. Other important hardware features which need to be exploited to the full are the memory-segmentation and virtual process control and implementation facilities. The implementation language chosen must take full advantage of these important features if the MULTUM ALP/3 architecture is to be used effectively.

2. Choice of Language

The outstanding difficulty in software development lies in the tracing and correction of errors. This is the major factor in determining the cost of large-scale software and on what can be accomplished in a given amount of time. Low-level languages emphasize the difficulty; high-level languages endeavour to minimize the difficulty. Economy of expression alone shortens the time taken to cut code and reduces errors significantly. Second in importance comes the problem of efficient object code. These high-level languages have traditionally been less satisfactory. This is because

- (i) coding has to be distorted to fit the language restrictions (e.g. FORTRAN!)

and (ii) data structures and loop control are not handled efficiently.

The choice lies between a low-level and a high-level language. There are five areas of choice

- (i) a macro-assembler
 - (ii) an intermediate-level language (that is, a language which permits data structure handling through user-specified code),
 - (iii) FORTRAN,
 - (iv) a known high-level language of proven effectiveness in implementing system software,
- and (v) a 'home-grown' language tailored to the task.

Experience suggests that (ii) and (v) are not to be considered further. Wirth's paper [1] on the implementation of PASCAL shows clearly why FORTRAN should not be considered further. In any case, the MULTUM architecture could not be effectively exploited in FORTRAN. The problem with case (v) is that even with a well-designed language which appears to epitomize the ultimate in language design (at least, to its designers), it is all-too possible to produce a language which does not perform as effectively as required. It is virtually impossible to produce a language which will be guaranteed to perform well, simply by thinking about it. SNOBOL is the classical example of a well thought-out language whose design turned out to be imperfect in the light of the way in which the language was actually used by most users. Further, the development of a home-grown language involves/

involves an even larger development time than if a ready-designed language is implemented. Discussion of this case, however is not completely closed by the preceding remarks.

The remaining possibilities are (i), (ii) and (iv). From these possibilities, the cost-effective choice has to be selected. The cost - criterion which is significant is the break-even cost per copy of the resultant software to the buyer. If this break-even cost is large then the seller may have to sell below this value, or accept a lower number of sales than appears in the formula. Either case means that the seller expects to make a loss overall. If the break-even cost is minimized then the likelihood of making a loss is minimized and (more positively) the likelihood of making a profit, both on software and on hardware linked with software, is maximized.

The break-even cost per copy of the software to be provided in a given software implementation language is given by the formula:

$$\frac{(\text{Cost in providing language})}{(\text{No of purchasers of language compiler})+1} + \frac{(\text{Cost to provide working software language})}{(\text{No of sales linked with provision of software})}.$$

In developing systems software, and in particular, in developing operating systems, the term 'cost to provide working software ' is by far the largest, and thus the language which minimizes the term without having a bad effect on the 'no of sales ' term, must be the one chosen.

The choice of language influences the 'no of sales ' term in several different ways. This term will be small if

- (a) delivery time is long,
 - (b) the jobs customers can run to their satisfaction taking advantage of the provided system (in the language) are substantially fewer in number and importance to them than the jobs the customers could be expected to run satisfactorily from an a-priori examination of the capabilities of the hardware.
- and (c) software modification to a given customer's specification is difficult.

The choice of the available macro-assembler for the implementation language is the choice which maximizes subsequent software development costs. This is due to the inherently poor security of assembly language and to the coding effort required to write software of a given level of complexity. The development effort is large and expensive and delivery time is long, both of which assist in pushing up the cost. The costly consequences of using assembly language are not offset sufficiently by macro facilities. Even very powerful macro-processors such as STAGE 2 and ML/1 have been described as unsuitable for initial software development by their designers for the reasons outlined.

The security argument and to a much lesser extent the amount of code to be written argument can again be used against the intermediate-level languages which can be made available relatively quickly (and at low 'cost in providing language') on the MULTUM or any other machine. (These languages suffer from the additional disadvantage of requiring the user to specify his data-structure/

structure handling operations in terms of the more basic operations of the language. At best the addressing modes of the MULTUM ALP/3 can only be exploited by building extensive optimization into the code generators of the compilers for these languages. This does not affect software development times, since code generation improvements can proceed in parallel with software development on an earlier version of the compiler, but does add to the cost. It is difficult to see with these languages how a completely satisfactory fit can be made between the built-in addressing modes and what the programmer is trying to get the machine to do. From the previous experience of others one can only expect that the object code will not be as efficient or as compact as it might be. Since people do not admit to their errors in public, as a rule, it is not easy to find references to case studies. One example near to home is the inability of the Kidsgrove Algol compiler to make good use of index registers on the KDF9 to translate for loops in ALGOL 60. This is at least in part a language design fault, although the provisions made in the KDF9 subset to make the optimization possible were wholly inadequate. Wirth [2] points out that a good match between the addressing mechanism in the machine and the data structures in the language is centrally important to the ultimate efficiency of the language, and this is the area where intermediate-level languages are weakest on the MULTUM ALP/3.

In summary, poor security lengthens delivery time and an imperfect match between coding and machine may reduce the effective usefulness of the machine to the user. In view of the alternatives, an intermediate-level language does not seem to be the best choice.

The remaining choice is of a bootstrappable or potentially available high-level language. The three main candidates PASCAL, ALGOL W and IMP. Of these, PASCAL is significantly the best choice. PASCAL is superior to the alternatives in the following respects

- (i) good, secure and extendable data types,
- (ii) efficient data-structure handling and loop control,
- (iii) unrivalled clarity and good economy of expression,
- and (iv) flexibility of expression (this is normal with most post-FORTRAN high-level languages).

Further the important ability to introduce new data types is unique to PASCAL. In addition, a PASCAL bootstrap exists and this has already been successfully used [3].

It should be possible to produce a particularly effective implementation of PASCAL on the MULTUM. Since PASCAL is a high-level language there is complete freedom in the way the data-structures are accrued and manipulated. Some pilot studies by Mr. Findlay of this department suggest that the code produced for the MULTUM ALP/3 by a good PASCAL compiler will be difficult to optimize further if revised by hand. In particular the addressing mechanisms of the MULTUM ALP/3 are used to great effect to produce particularly compact code. Further, one of the most difficult data structures to implement is the POWERSET. A good (and minimally restricted!) implementation of powerset is a particularly powerful tool in systems programming, and the instructions peculiar to/

peculiar to the MULTUM may be exploited to provide the best implementation of this construct made available so far.

In summary, the preceding argument indicates that the cost-effective choice is of the high-level language PASCAL. In the introduction to this section it was suggested that high-level languages tended to yield less efficient (and compact) object code. Four points are worth making in connection with PASCAL in this context. These are that

- (i) algorithmic rather than coding efficiency is more important in systems programming,
- (ii) the quantity of coding to be written affects the efficiency to which algorithms are coded, and in large-scale systems coded in assembly language this is very poor,
- (iii) PASCAL offers better coding efficiency than the alternatives examined for MULTUM ALP/3,
- and (iv) system efficiency can often be greatly improved by modest re-coding in small initial areas.

Finally it is well-known that certain manufacturers systems are almost unbelievably inefficient in their use of the machine. This is known to be due to the sheer magnitude of the software effort mounted. Any high-level language with adequate data structures and tolerable coding efficiency would appear to be preferable, even in venues of the ultimate efficiency of the object program, to coding large-scale systems software in assembly language or something not far removed from assembly language. PASCAL is significantly better than what would be required to effect a minimal improvement on this situation. Additionally the use of a high-level language, and PASCAL in particular, will greatly improve the reliability of the written software, which is a major factor contributing to system development costs.

3. MULTUM implementation of PASCAL

A first estimate of the effort needed to bootstrap PASCAL onto the MULTUM is six to eight man-months. This should be sufficient to provide a compiler which produces good code. An efficient PASCAL compiler is only necessary at a late stage in software development and a PASCAL compiler which produces inefficient code but only requires minimal reconstruction of an existing code generator could be produced in considerably less time (say, three man-months). Further, most of this development can be done on any machine which is currently host to the PASCAL compiler. This note proposes three stages of development

- (i) the provision of a minimally reconstructed PASCAL compiler as soon as possible. (This could pre-date the delivery of the MULTUM with ALP3 at Glasgow!)
- (ii) the reconstruction of the code generator to produce the best possible code for the MULTUM (the compiler should occupy around 36k of store at this point)
- and (iii) the development of both the language and the compiler to provide the most extensive program development aids we can, and to add the synchronizing facilities desirable for operating system work.

The remainder of this section covers the sorts of additional facilities which are highly desirable for operating system development. The facilities come under three headings. These are

- (i) compiler modifications which do not affect the language definition
- (ii) checking facilities which other compilers may treat as commentary
- (iii) synchronization facilities.
- and (iv) assembly code routines or modules.

In category (i) come (a) the calculation at compile-time of all expressions involving constants, and subsequent treatment of these expressions as constants, and (b) the use of constants in if and case statements to act as code selectors.

In category (ii) come assertions, module specifications and module interface specifications. Modules should be treated as a form of macro by the user and as a domain of store by the compiler.

The synchronization facilities suggested are exactly those suggested by Hansen [4] after Hoare. These are much easier to use than Dijkstra Sempahores, although it is uncertain whether they will normally be noticeably less efficient or not. In the limiting case they can be used just like Dijkstra Semaphores with the same efficiency.

In addition to these built-in features, a macro-processor should be provided as a pre-pass to compiling. The precise choice of macro-processor is relatively unimportant, although a macro-processor tailored to the PASCAL implementation allowing a distributed macro name and parameter checking gives a powerful language extension facility.

For situations of last resort, assembly code routines or modules should be provided. These may be needed to 'tune' frequently used system software components.

4. Note

This note is intended to apply only to MULTUM ALP/3 processor configurations. PASCAL could be provided on an ALP/2 provided that the floating-point operations of the ALP/3 were simulated. An ALP/1 version is possible, but the reduced number of addressing modes invalidates a good deal of the argument given in section 2. Additionally, the semantics of the language may have to be altered to fit the ALP/1. It is definitely NOT advisable to standardize the semantics OR the code generator to an ALP/1 version. It is not our intention to provide a PASCAL compiler for the ALP/1; ICS may however wish to take this up themselves.

5. References

- [1] N. Wirth. The Programming Language PASCAL. Acta Informatica 1 35-63 (1971).
- [2] N. Wirth. The design of a PASCAL compiler Software Practice and Experience Vol.1 309-333 (1971).
- [3] J. Welsh and C. Quinn: A PASCAL Compiler for ICL 1900 Series Computers. Software Practice & Experience, Vol. 2 73-77 (1972).
- [4] P. Hansen. A Comparison of Two Synchronizing Concepts Acta Informatica 1 190-199 (1972).